Models of Computation

- In order to define classes like P precisely, we need to specify our model of computation
- In discussing the time complexity of algorithms considered earlier in the course, we have sometimes implicitly assumed the "sequential RAM" model
  - For input size n, we assume a "word size" of $O(\log n)$ bits
  - We can perform ALU operations on word-sized operands in constant time
  - We can read/write the word of memory stored at the location given by an $O(\log n)$-bit index in constant time
- In complexity theory it is generally more convenient to work with the simpler Turing machine model

Turing Machines: Overview

- The high-level objective is to define a simple machine model that is general enough to be able to simulate any "feasible" model of computation
- Ideally, we would like this simulation to be "efficient"
  - For example, it turns out that any program designed for the sequential RAM model can be simulated by a Turing machine with polynomial slowdown
  - Thus it makes no difference whether we define the complexity class P (or the notion of a polynomial-time reduction, NP, et cetera) with respect to the sequential RAM model or the Turing machine model
- Next we will formalize our notion of a (deterministic) Turing machine
  - Many minor variations of our definition are possible

The Main Components of a Turing Machine

- A one-way infinite tape partitioned into cells, each of which can store a single symbol drawn from some finite alphabet
- A head which at any point in the computation is positioned over some cell of the tape
- A finite control that dictates how the state (tape contents, head position) is updated at any given step of the computation

The Finite Control

- Finite number of states, including an initial state and a halt state
- A transition function that specifies the following information for every possible combination of the current state and currently scanned symbol
  - Whether to (1) move the head left, (2) move the head right, or (3) overwrite the current cell with a specified symbol
  - The new state

Input Convention

- The finite alphabet associated with the Turing machine includes a special blank symbol
- The input is a string over this alphabet that does not include any blanks
- If the input string is of length k, it is provided in the first k cells of the tape
- The rest of the tape is filled with blanks

Output Convention

- For an execution of a Turing machine to be considered to terminate properly, we require the following
  - The machine is in the halt state with read-write head scanning leftmost cell
  - The entire tape is blank except for some non-blank prefix, which represents the output string
  - For a decision problem, we require that the output string is either 1 (“yes”) or 0 (“no”)

Categories of Turing Machine Executions

- What different kinds of things can happen when a Turing machine M is executed on a given input string w?
- One possibility is that the execution terminates properly after some finite number of steps and produces an output string (or output bit, in the case of a decision problem)
- Another possibility is that the execution terminates improperly after some finite number of steps
  - This category includes executions where the read-write head is moved off the left end of the tape
- Still another possibility is that the execution "hangs", i.e., it never terminates

Turing-Computable Functions

- Let Σ denote the set of non-blank symbols in the alphabet associated with a Turing machine M
- We say that M computes the function f from Σ^* to Σ^* if, for any input string w in Σ^*, when we execute M on input w, the execution terminates properly with output $f(w)$

Decision Problems as Languages

- Fix a finite alphabet Σ, such as $\{0, 1\}$
- A language over Σ is a subset of Σ^*, the set of all finite-length strings with symbols drawn from Σ
- Sometimes it is useful to think of a decision problem as a language
  - Assume the input to the decision problem is a string in Σ^*
  - The language associated with the decision problem is the set of all input strings w for which the output is "yes"

Turing-Decidability

- A language L is decided by a Turing machine M if the following conditions hold
  - For any input string w in L, when we execute M on input w, the execution terminates properly and produces output 1
  - For any input string w not in L, when we execute M on input w, the execution terminates properly and produces output 0

Church-Turing Thesis

- The Church-Turing thesis posits that any function that can be feasibly computed can be computed by a Turing machine
- As we have noted earlier, a Turing machine can simulate any algorithm for the sequential RAM model
  - In fact, it can do so with only polynomial slowdown

Complexity-Theoretic Church-Turing Thesis

- A stronger version of the Church-Turing thesis posits that any feasible model of computation can be simulated by a Turing machine with polynomial slowdown
- Seems to hold for classical (i.e., not based on quantum mechanics), deterministic models of computation
- What if we allow randomization, or computational models based on quantum mechanics?
  - It remains open whether these models can be simulated by classical deterministic Turing machines in polynomial time

The Class P

- We define the class P as the set of all languages decided by a deterministic Turing machine in polynomial time
  - That is, there exists a positive constant c such that the number of steps taken to determine whether a given string x is in the language is $O(|x|^c)$
- Equivalently, the class P is the set of all decision problems solvable in polynomial time in the sequential RAM model

Examples of Problems in P

- While we have presented many polynomial-time algorithms in the class, most of the associated problems are not in P because they are not decision problems
- For example, the maximum flow problem is not a decision problem since the output is not yes/no
- For any such maximization (or minimization) problem, there is a natural decision variant
  - Given a network flow instance G and a bound B, does G admit a flow with value at least B?
  - Given a knapsack instance I and a bound B, is it possible to fit items worth B or more into the knapsack?

Remarks on the Restriction to Decision Problems

- It is generally easy to argue that existence of a polynomial-time algorithm for the decision version of a problem implies a polynomial-time algorithm for the original version
- For example, how could you use a polynomial-time algorithm for the decision version of the knapsack problem to solve the knapsack problem?
  - Assume that the item weights and values are integers
- Thus, to determine whether the knapsack problem is polynomial-time solvable, it suffices to determine whether the decision version of the knapsack problem belongs to P
- Similarly, for most other optimization problems there is no loss of generality in focusing on the decision version

The Class NP: Original Definition

- Traditionally, the class NP is defined in terms of nondeterministic Turing machines
  - The finite control consists of a transition relation rather than a transition function
  - In other words, more than one transition might be admissible in a given state
- We the define the notion of a nondeterministic Turing machine M accepting (as opposed to deciding) a given language L
  - Given an input string that does not belong L, no execution of M leads to an accepting state
  - Given an input string that does belong to L, there exists an execution of M leads to an accepting state
- The class NP is the set of all languages accepted by a nondeterministic Turing machine in polynomial time

The Class NP: An Equivalent Definition

- A language L belongs to NP if there exists a positive constant c and a polynomial-time algorithm $\mathcal{A}(x, y)$ such that the following conditions hold
  - If x belongs to L, then there exists a y such that $|y| = O(|x|^c)$ and $\mathcal{A}(x, y)$ outputs 1
  - If x does not belong to L, then $\mathcal{A}(x, y)$ outputs 0 for all y
- We will work with the above definition, which is easily shown to be equivalent to the original
- In applications of the above definition, the algorithm $\mathcal{A}$ is sometimes referred to as the "verifier", and the string y is sometimes referred to as a (purported) "short certificate" that x belongs to L

Example: The Decision Version of Knapsack

- We now argue that the decision version of knapsack (KNAPSACK) belongs to NP
- Design a polynomial-time verifier algorithm $\mathcal{A}(x, y)$ for KNAPSACK
  - Given a positive instance x, what is a short certificate y that causes $\mathcal{A}(x, y)$ to output 1?
  - Given a negative instance x, how can we be sure that $\mathcal{A}(x, y)$ outputs 0 for all y?
- We conclude that KNAPSACK belongs to NP

Example: Satisfiability of Propositional Formulas

- The satisfiability problem (SAT) plays a central role in complexity theory
  - Let f be a given propositional formula using the connectives $\wedge$, $\vee$, and $\neg$, say
  - We say that f is "satisfiable" if there exists a truth assignment to the variables of f under which f evaluates to true
- Let f be a positive satisfiability instance
- It is easy to design a polynomial-time verifier for the decision version of knapsack
  - We conclude that SAT belongs to NP

Polynomial-Time Reductions

- We say that a computational problem X is polynomial-time reducible to a computational problem Y, written $X \leq_P Y$, if an arbitrary instance of X can be solved using (1) a polynomial number of standard computation steps and (2) a polynomial number of calls to a black box that solves Y
  - Note: The bound for (1) includes the cost of writing (resp., reading) the input (resp., output) for each call to the black box
- We have seen a lot of examples of such polynomial-time reductions in the course
  - The maximum-cardinality bipartite matching problem is polynomial-time reducible to the maximum flow problem
  - The maximum flow problem is polynomial-time reducible to the linear programming problem
  - The optimization problem of knapsack is polynomial-time reducible to the decision version

Using Polynomial-Time Reductions to Establish Upper Bounds

- Let X and Y be two computational problems, and assume that Y is polynomial-time solvable
- If we show that $X \leq_P Y$, we can conclude that X is polynomial-time solvable
- Thus polynomial-time reductions are a useful tool for growing the class of computational problems that are known to be "easy" (i.e., polynomial-time solvable)

Using Polynomial-Time Reductions to Show Hardness

- Let X and Y be two computational problems, and assume that X is suspected to be "hard" (i.e., not polynomial-time solvable)
- If we show that $X \leq_{\mathrm{P}} Y$, we can conclude that Y is likely to be hard

A Potential Pitfall: Reducing in the Wrong Direction

- To avoid reducing in the wrong direction, it may be useful to think of all polynomial-time solvable problems as having "low" hardness, and all other problems as having "high" hardness
  - Thus $X \leq_P Y$ if and only if hardness$(X) \leq$ hardness(Y)
  - Showing $X \leq_P Y$ rules out the possibility that X is hard and Y is not
  - If we want to give evidence that a problem Y is hard, we should expect to provide a reduction of the form $X \leq_P Y$ where X is (believed to be) hard

The 3-SAT Problem

- The 3-SAT problem is the special case of SAT in which the following conditions hold
  - The input formula is the conjunction (AND), of a number of "clauses"
  - Each clause is the disjunction (OR) of exactly three "literals"
  - Each literal is either a variable or the negation of a variable
  - The three literals in any clause are associated with distinct variables
- A formula satisfying the above conditions is said to be in 3-CNF form
  - The abbreviation CNF stands for "conjunctive normal form"

Sketch of a Reduction from SAT to 3-SAT

- Let f be a given SAT instance
- Let T be the parse tree corresponding to f
- By introducing an additional variable for each internal node of T, we can easily write down a propositional formula f' satisfying the following conditions
  - The size of f' is polynomial (linear, in fact) in the size of f
  - The formula f is satisfiable if and only if f' is satisfiable
  - The formula f' is the conjunction of a number of clauses
  - Each clause of f' has one of the following forms, where a, b, and c denote literals (associated with distinct variables): $a \leftrightarrow b \wedge c$; $a \leftrightarrow b \vee c$; $a \leftrightarrow \neg b$; a

Sketch of a Reduction from SAT to 3-SAT (cont'd)

- To complete the reduction, we show how to rewrite each of the four kinds of clauses as a logically equivalent 3-CNF formula of constant size
- By replacing each clause of f' with such a logically equivalent 3-CNF formula, we obtain a 3-CNF formula f'' that is logically equivalent to f'
- Thus f'' is satisfiable if and only if f is satisfiable
- We conclude that SAT $\leq_{\mathrm{P}}$ 3-SAT